# *Module 01: Elliptic Curve Cryptography*

## *Week-02: Implementing ECC and ECDSA*

*Jan Gilcher, Kenny Paterson, and Fernando Virdia*

*jan.gilcher@inf.ethz.ch, kenny.paterson@inf.ethz.ch,*
*fernando.virdia@inf.ethz.ch*

*September 2021*

Hi all! Welcome to the first Information Security Lab for this semester. This lab is part of the Module-01 on *Elliptic Curve Cryptography*, or ECC in short. Due to technical limitations the labs will only take place in presence. You are also welcome to post your questions and to join ongoing discussions on the Moodle forum at the following link:

`https://moodle-app2.let.ethz.ch/course/view.php?id=15448`

Students who cannot attend the labs in presence are especially encouraged to use Moodle to ask questions.

### *Overview*

This lab is focused on implementing a framework for Elliptic-Curve Cryptography (ECC) from the ground-up, and using the same to build an implementation of the Elliptic Curve Digital Signature Algorithm (ECDSA) [1]. You have already seen the theoretical descriptions of the relevant ECC operations, as well as ECDSA, during the lectures; but we will recap the necessary material in this lab sheet to help with the implementation.

Please note that we expect you to use Python-3 for this lab. We will not accept submissions/solutions coded in any other language, including older versions of Python. *So please make sure that your submissions are Python-3 compatible.*

### *Getting Started*

You are provided with a skeleton file named `module_1_ECC_ECDSA_Skel.py`. This file has several unimplemented functions related to basic ECC operations as well as ECDSA that you are expected to implement. For your benefit, some low-level functions are already implemented for you. These include:

- `egcd`: Computes the gcd of two integers using the Euclidean algorithm.
- `mod_inv`: Computes $a^{-1} \mod p$ for input integers $a$ and $p$.

- `hash_message_to_bits`: Outputs a bit string representing the SHA-256 hash of an input message msg.
- `bits_to_int`: Truncates and converts a bit string into an integer modulo $q$, as described in the lecture.

### *class Curve*

The skeleton file also has an implementation of `class Curve` - a Python class abstracting an elliptic curve object. In this lab, we will assume that any elliptic curve is represented in short Weierstrass form [2,3]. Accordingly, any `Curve` object has the following attributes, all of which are represented in Python as (big) integers:

- `(a, b)`: The coefficients in the short Weierstrass representation.
- `p`: The order for the base field $\mathbb{F}_p$.
- `(P_x, P_y)`: The coordinates for the base point $P$ on the curve.
- `q`: The order of the additive group of points generated by the base point $P$.

You can initialize a `Curve` object by invoking the following Python statement:

```
curve = Curve(a, b, p, P_x, P_y, q)
```

The `class Curve` also has certain functions that are already implemented in the skeleton file. These include functions to check if a curve is singular, to check if a point lies on the curve, and if two curve objects are equal. Some of these functions are for your inspection, while some others are used subsequently.

### *class PointInf*

The skeleton file has an (incomplete) implementation of `class PointInf` - a Python class abstracting a *point at infinity* object. Recall that the point of identity is essentially the additive identity for the group of points generated by the base point $P$. We represent it using a different class as compared to a normal point for ease of implementation.

A `PointInf` object only as a single attribute, namely a `Curve` object depicting the curve it lies on. You can initialize a `PointInf` object by invoking the following Python statement:

```
pointinf = PointInf(curve),
```

where `curve` is a `Curve` object initialized as described earlier.

The `PointInf` class also has three unimplemented functions, out of which you are expected to implement two:

- `double`: This function should return the outcome of doubling a `PointInf` object (what should the return type of the function be?).
- `add`: This function should return the outcome of adding a `PointInf` object with either another `PointInf` object or a `Point` object (described subsequently). Note that depending on the nature of the operand, this function should return either a `PointInf` object or a `Point` object.
- `negate`: This function should return the outcome of negating a `PointInf` object (what should the return type of the function be?). Implementing this function is *optional* and is *not evaluated*.

### *class Point*

As you might have guessed by now, the skeleton file also has an (incomplete) implementation of `class Point` - a Python class abstracting any point on the curve other than the point at infinity. In this lab, we represent any point on an elliptic curve using *affine coordinates*. Accordingly, any `Point` object has the following attributes:

- `curve`: A `Curve` object depicting the curve the point lies on.
- `(x, y)`: The x and y coordinates of the point (again represented as big integers).
- `p`: The order for the base field $\mathbb{F}_p$.

Note that the attribute `p` is also indirectly available as a sub-attribute of the `curve` object attribute, but we store it explicitly to avoid long indirect references (trust us, it makes life easier).

You can initialize a `Point` object by invoking the following Python statement:

```
point = Point(curve, x, y),
```

where `curve` is a `Curve` object initialized as described earlier.

The `Point` class again has five unimplemented functions, out of which you are expected to implement three:

- `double`: This function should return the outcome of doubling a `Point` object. Recall from the lecture that given $P = (x, y)$, we have $[2]P = (x', y')$ computed

using the following equations:

$$\begin{aligned} \lambda &= (3x^2 + a)/2y \mod p. \\ x' &= (\lambda^2 - 2x) \mod p. \\ y' &= -(y + \lambda \cdot (x' - x)) \mod p. \end{aligned}$$

- `add`: This function should return the outcome of adding a `Point` object with either a `PointInf` object or a `Point` object. In particular, you should check the input `other` to see which kind of object it is.

  Recall from the lecture that given $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, we have $P_1 + P_2 = (x', y')$ computed using the following equations:

$$\begin{aligned} \lambda &= (y_1 - y_2)/(x_1 - x_2) \mod p. \\ x' &= (\lambda^2 - x_1 - x_2) \mod p. \\ y' &= -(y_1 + \lambda \cdot (x' - x_1)) \mod p. \end{aligned}$$

  **Note:** Your implementation should handle the special cases when:

  - $(x_1, y_1) = (x_2, y_2)$,
  - $x_1 = x_2$ and $y_1 \neq y_2$.

  In the second case, what should the outcome of the `add` function be and how should this affect its return type?

- `negate`: This function should return the outcome of negating a `Point` object. Implementing this function is *optional* and is *not evaluated*.

- `scalar_multiply`: This function should return the outcome of scalar multiplication of between `Point` object and an input `scalar` (represented as a big integer). The implementation need not be constant-time with respect to the bits of `scalar`. In particular, you can use the *double-and-add* algorithm by invoking the `double` and `add` functions that you have implemented earlier.

  **Note:** Note that the outcome of the `scalar_multiply` function could be the point at infinity (think when). Your implementation should be flexible enough to accordingly return a `PointInf` object in this case. In all other cases, the function should return a `Point` object.

- `scalar_multiply_scalar_Montgomery_Ladder`: This function should return the outcome of scalar multiplication of between `Point` object and an input `scalar` (represented as a big integer). The implementation should *be* constant-time with respect to the bits of `scalar`. In particular, we advocate using the well-known *Montgomery Ladder* algorithm [4].

  **Note:** Implementing this function is *optional* and is *not evaluated*. However, we encourage you to implement the Montgomery ladder algorithm, and test that your implementation is indeed constant-time using a simple timer module built into Python. If in the future you ever develop an ECC library of your own, you should always make sure to implement a constant-time realization of scalar multiplication to resist potential leakages that can be exploited to devastating effect by *side-channel attacks*.

## *ECDSA*

The last part of today's lab focuses on ECDSA - a digital signature standard based on ECC. The skeleton file provides you with an implementation of `class ECDSA_Params` - a Python class abstracting the public parameters for any ECDSA scheme. Any `ECDSA_Params` object has the following attributes:

- `curve`: A `Curve` object depicting the curve to be used by the implementation.
- `p`: The order for the base field $\mathbb{F}_p$.
- `P`: A `Point` object depicting the base point $P$ on the curve.
- `q`: The order of the additive group of points generated by the base point $P$.

You can initialize an `ECDSA_Params` object by invoking the following Python statement (very similar to the Python statement for initializing a `Curve` object):

```
params = ECDSA_Params(a, b, p, P_x, P_y, q)
```

Next, we have three unimplemented ECDSA functions which you are expected to implement:

- `KeyGen`: This function takes as input `params` - an `ECDSA_Params` object, and outputs a key pair $(x, Q)$ where $x$ is the private signing key and $Q$ is the public verification key. Recall from the lecture notes that we have:

  $$x \xleftarrow{R} [1, q-1] \quad , \quad Q = [x]P.$$

  The function should output the tuple $(x, Q)$ where $x$ should be represented as a (big) integer and $Q$ should be represented as a `Point` object.

- `Sign_FixedNonce`: This deterministic function takes as input:
  - `params` - an `ECDSA_Params` object,
  - a nonce $k \in [1, q-1]$,
  - a private signing key $x \in [1, q-1]$,
  - a message `msg`,

  and outputs a signature $(r, s) \in [1, q-1] \times [1, q-1]$ (represented as a pair of big integers) where:

  $$h = \mathcal{H}(\mathsf{msg}) \quad , \quad P' = [k]P \quad , \quad r = \text{x-coord}(P') \mod q \quad , \quad s = k^{-1} \cdot (h + x \cdot r)) \mod q,$$

  where $\mathcal{H}$ is a function that hashes `msg` to a bit string, truncates it and finally maps it to an integer modulo $q$.

  **Note:** If you pause to think for a couple of seconds, you will realize that the skeleton file already provides you with the functions needed to implement $\mathcal{H}$. For reasons concerning evaluation, we ask you to use *this* implementation of $\mathcal{H}$ and *not* your own.

- `Sign`: This function is simply the randomized variant of `Sign_FixedNonce` in that it should sample the nonce $k$ uniformly from $[1, q-1]$. More specifically, this function takes as input:
  - `params` - an `ECDSA_Params` object,
  - a private signing key $x \in [1, q-1]$,
  - a message `msg`,

  and outputs a signature $(r, s) \in [1, q-1] \times [1, q-1]$ (represented as a pair of big integers).
- `Verify`: This deterministic function takes as input:
  - `params` - an `ECDSA_Params` object,
  - a public verification key $Q$ - a `Point` object,
  - a message `msg`,
  - a signature $(r, s)$ (represented as a pair of big integers)

  and outputs either 0 or 1. Recall from the lecture notes that letting:

  $$h = \mathcal{H}(\mathsf{msg}) \quad , \quad w = s^{-1} \mod q \quad , \quad u_1 = w \cdot h \mod q \quad , \quad u_2 = w \cdot r \mod q,$$

  and letting

  $$Z = [u_1]P + [u_2]Q := (x_Z, y_Z),$$

  we have:

  $$\texttt{Verify}(\texttt{params}, Q, \mathsf{msg}, (r,s)) = \begin{cases} 1 & \text{if } (r,s) \in [1,q-1] \times [1,q-1] \textit{ and } r = x_Z \mod q, \\ 0 & \text{otherwise.} \end{cases}$$

### *Testing and Evaluation*

Together with the skeleton file, we have provided a test script with the necessary parameters to create an `ECDSA_Params` object representing the NIST P-256 curve [5]. We have also provided you with a sample instantiation for the same. Subsequently, you are provided with five test modules - four of which involve file reads and writes. You can use these modules to test your implementation.

To run the tests, you need to make the following function call to `run_tests`. This code is already included at the end of the skeleton file.

```
from module_1_ECC_ECDSA_tests import run_tests
run_tests(ECDSA_Params, Point, KeyGen, Sign, Sign_FixedNonce, Verify)
```

**Note:** You are free to test your code using your own custom-designed test modules. However, we will be using the *same* test modules as in the skeleton file to evaluate your submissions under an automated evaluation framework. Your final submission should have these test modules as is. *Please do not tamper with the test modules in any way to avoid interfering with our automated evaluation frameworks.*

### *Testing and Evaluation of Deterministic Functions*

The first four test modules involve file reads and writes and are designed to test the following deterministic functions: `scalar_multiply`, `add`, `Sign_FixedNonce` and `Verify`. For each of these test modules, you are provided with the input test vectors and the corresponding output vectors in separate input and output files. When you execute each test module, you will generate an output file. Your implementation is correct if the contents of this file *exactly* match the contents of the output file provided to you. This check is performed automatically by the tests modules, and a warning is raised if the check fails.

When we evaluate your submissions, we will run the exact same test modules, albeit with respect to our own privately generated input and output files, which will not be made public prior to evaluation. This is why we ask you to leave the test module codes as is in your final submissions. *Failure to adhere to this instruction will incur heavy penalties*.

### *Testing and Evaluation of ECDSA Correctness*

The final test module will evaluate the correctness of your overall ECDSA implementation. It will generate multiple random signatures (using your implementation of `Sign`) under many different key pairs (generated using your implementation of `KeyGen`), and verify each of those signatures (using your implementation of `Verify`). If any of the signatures fail to verify, the test will throw an exception, indicating failure.

We will evaluate your submissions using exactly the same test. *Any attempt to tamper with the test code will also incur similarly heavy penalties*.

**Summary of Evaluation Criteria.** To summarize, you will be evaluated based on the correctness of your implementation of the individual functions:

1. `scalar_multiply` (6 points)
2. `add` (6 points)
3. `Sign_FixedNonce` (6 points)
4. `Verify` (6 points)

as well as the overall correctness of your ECDSA implementation (6 points). (So the maximum score for this lab is 30 points; next week's lab is worth 70 points.)

### *Submission Format*

Your completed submission for this week should consist of a *single* Python file, and should be named "`module_1_ECC_ECDSA.py`".

You are expected to upload your submission to Moodle. The submissions for weeks 2 and 3 should be bundled into a single archive file named "module_1_submission_[insert LegiNo].zip". Submission instructions for the solution to week 3 will be provided separately in the lab sheet for week 3.

In conclusion, *happy coding*!

## *References*


1. *Public Key Cryptography For The Financial Services Industry: Agreement Of Symmetric Keys Using Discrete Logarithm Cryptography.* ANSI X9.42-2003 (2003). `https://webstore.ansi.org/standards/ascx9/ansix9422003r2013`
2. *Math. Werke. Bd. 1–2.* Weierstrass, K. (1894–1895)
3. *Elliptic functions.* Lang, S. (1987). `https://link.springer.com/content/pdf/10.1007%2F978-1-4612-4752-4.pdf`
4. *Montgomery curves and the Montgomery ladder.* Bernstein, D. J., & Lange, T. (2017). IACR Cryptol. ePrint Arch., 2017, 293. `https://eprint.iacr.org/2017/293.pdf`
5. *Elliptic Curve Cryptography Subject Public Key Information.* Turner *et al.* (2009). `https://www.ietf.org/rfc/rfc5480.txt`